Hindawi
Security and Communication Networks
Volume 2017, Article ID 4121765, 13 pages
https://doi.org/10.1155/2017/4121765

Research Article

Android Rooting: An Arms Race between Evasion and Detection

Long Nguyen-Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung

School of Electronic Engineering, Soongsil University, Seoul, Republic of Korea

Correspondence should be addressed to Souhwan Jung; souhwanj@ssu.ac.kr

Received 1 May 2017; Revised 31 July 2017; Accepted 16 August 2017; Published 29 October 2017

Academic Editor: Zonghua Zhang

We present an arms race between rooting detection and rooting evasion. We investigate different methods to detect rooted device at both Java and native level and evaluate the counterattack from major hooking tools. To this end, an extensive study of Android rooting has been conducted, which includes the techniques to root the device and make it invisible to the detection of mobile antimalware product. We then analyze the evasion loopholes and in turn enhance our rooting detection tool. We also apply evasion techniques on rooted device and compare our work with 92 popular root checking applications and 18 banking and finance applications. Results show that most of them do not suffice and can be evaded through API hooking or static file renaming. Furthermore, over 28000 Android applications have been analyzed and evaluated in order to diagnose the characteristics of rooting in recent years. Our study shows that rooting has become more and more prevalent as an inevitable trend, and it raises big security concerns regarding detection and evasion. As a proof of concept, we have published our rooting detection application to Google Play Store to demonstrate the work presented in this paper.

1. Introduction

Android has been dominating in the mobile market for 7 years consecutively and this OS will continue doing so. According to IDC report in August, 2016 [1], Android has led the smart phone market share over the last 4 quarters with the highest share of 87.6% in Q2. While Google code change will not be made available to public immediately, new version release still includes most of what community needs: factory images, source code, OTA distribution channels, and APIs. Openness is inarguably the main reason why Android OS gains its popularity quickly.

Android is considered as a very complex ecosystem; each device is a composition of different software (open source and closed source), different hardware (screen sizes, manufacturers), and different distributors. That makes the exploit universally impossible but also makes the job of auditing difficult as reviewing every device and their software is a huge amount of work. The process of a security update for a specific Android device can be summarized as follows: (1) security flaw found in an OS, (2) Google release the patch, and (3) OEMs (Original Equipment Manufacturers) adopting the patch and including it in their custom built devices. The whole process can take months to eventually fix the vulnerability. This fragmentation nature of Android makes this operating system a fruitful land for attackers [2].

Even though Android devices come with many customizations, they are still limited within some sets of APIs and custom builds from OEMs [3, 4]. Many advanced users often tweak their devices by gaining access to the highest privilege, also known as root or super user, and are able to manipulate the OS to any custom build of their choices. There are plenty of benefits that encourage users to root their devices [5], some important reasons could be mentioned as follows: (1) getting rid of bloatware (software that are preinstalled by vendors), (2) performing full system backup (unrooted devices only allow backup of userland data), (3) updating the OS in advance before OEMs release (including new features and especially security patches), and (4) acquiring new functionalities brought by custom ROMs, as well as making use of rooting-related applications.

According to Google Report on Android Security since 2014 [6–8], the term "rooting" is frequently mentioned as one of the most important factors in privilege escalation threats. Rooting applications was added to PHAs (Potentially Harmful Applications). Despite being classified as "harmful,"

these applications are not considered "malicious" as long as their behaviors are adequately disclosed to the user. As shown in the reports, different tools have been used by Google to evaluate rooting. In 2014 report, Google Verify Apps identified rooting applications installed on 0.29% of devices (outside Google Play), and fraction of devices with a PHA Installed is doubled with rooting applications included. In 2015 report, Google employed Anomaly Correlation Engine to detect rooting applications and other PHAs, resulting in 0.32% over 1 billion devices that had rooting applications installed (page 36). In 2016, Google Attestation served 25% more than 2015, with nearly 200 million requests per day; user-intended rooting installations comprise 0.346% of all installs. As Verify Apps, Anomaly Correlation Engine, and Attestation all belong to Google SafetyNet APIs, we will discuss the internals of this Google product in the next sections, as well as some studies regarding rooting in Android.

In this work, we use different terms to denote rooting-related applications.

(1) One-Click Rooting Apps (or Just Rooting Apps). They are applications that are used for rooting purpose. By exploiting the flaws of the operating system to perform privilege escalation, these apps are strictly prohibited by Google in Play Store and classified as PHAs (Potentially Harmful Applications).

(2) Root Manager Apps. They are applications that act as secure gateways to root privilege. One of the most well-known root managers is SuperSU by chainfire [9].

(3) Root Apps. They are applications that use root privilege to extend their features in the device. This term is not to be confused with one-click rooting apps.

(4) Root Checking Apps. They are applications that check for the signs of rooting. In general, they can be modules embedded inside finance applications or antimalware products.

Due to historical reason, users and developers keep using the terms "root apps" and "rooting apps" to imply (1), (3), and even (2) interchangeably. We will try our best to avoid these terms and differentiate the apps throughout the paper by providing explanations in context.

Rooting has been a controversial topic in mobile world since 2009 and back to the birth of Unix Operating System [10]. The act of rooting stands on the boundary between user experience and device security. Rooting evasion allows applications to run on rooted devices, despite not being supposed to do so for security reason. While banking and security companies are trying to harden their root checking modules, a large community, however, is doing the opposite [11]. The strong support on both parties makes it difficult to put an end to this controversy. Being inspired by an arms race between rooting and antirooting, we conducted several studies regarding the Android OS. Starting from "how rooting works" and "what are the risks of rooting" to "why Google and banking, finance companies frequently raise concerns about this topic," we try to understand current situation of evasion and detection techniques and then revisit the work of rooting detection in Android 4.4 and later versions. We also perform a large-scale analysis on different datasets of Android applications in order to have a better understanding of the characteristics of Android applications in recent years.

Our contributions in this study can be listed as follows.

(1) We investigate current rooting methods for Android devices (Section 2.2) and the techniques to avoid being detected by banking applications or antimalware product, targeting Android version 4.4 and later ones.

We conduct a survey of 110 root checking applications (92 root checking apps and 18 banking/finance apps) and reveal their current implementation against rooting. While 89 root checking apps can be evaded through Java API hooking, we discover 18 banking/finance apps that implemented native code check and therefore are immune to this evasion technique. We then try to customize the current system and successfully make 10 out of 18 apps run in rooted environment. This effort helps us confirm that, with the same approach, any root checking application can be evaded as it is just a matter of time.

(2) Based on characteristics of Android OS and behaviors of rooting applications, we collect a set of feasible APIs and embedded commands and implement an Android application to carry out the work of rooting detection and construct it as an Android app, which we have published to Google Play Store recently (S4URC Root Checker in Figure 5) [12].
(3) We perform a large-scale analysis over 28000 Android applications to evaluate other aspects of this study, which includes 7200 apps from Play Store and more than 21000 malware types we have collected from different websites. These apps are decompiled and the source code was analyzed to reveal their behaviors regarding the rooting. This work helps us have a better understanding of malware behaviors, as well as the ability to measure the prevalence of privilege escalation through rooting in Android. The detailed analyses will be discussed in Section 6.

The rest of this paper can be summarized as follows: Section 2 briefly provides some background regarding rooting and privilege escalation. Section 3 presents related works of Android rooting, as well as security concern and existing solutions in paperwork. Sections 4 and 5 discuss techniques to bypass rooting check of popular applications and implementation of our root checking application [12]. Analysis results will be provided in Section 6, and we conclude our work in Section 7.

2. Background

In this section, we provide some background of the Android Security Architecture, rooting mechanisms, and techniques for hooking Android APIs.

2.1. Android Security Architecture. Android OS are designed by stacking different software components on top of each other and secure them from the ground up; each component assumes that the components below are properly secured. Most code and daemons are running as normal privilege with the restriction of the application sandbox, as opposed to root privilege (or superuser privilege) which is only granted

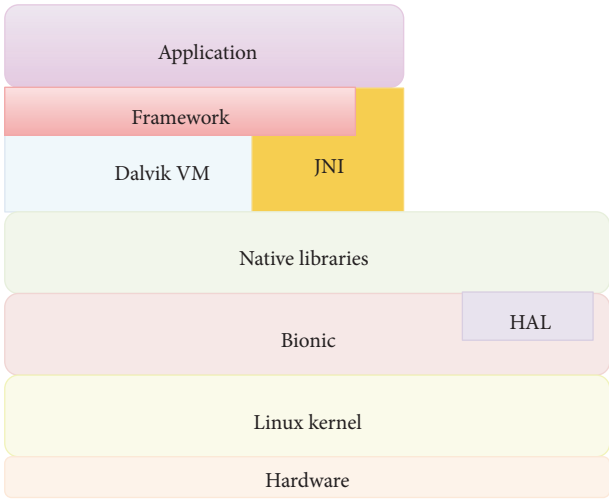

Figure 1: Android architecture.

to initial processes for firing up the device. The stacking of security components corresponding to each layer is described in Figure 1. Further reference regarding the design of Android OS can be found in the official Android page [13].

2.2. Principles of Rooting. Rooting, in smartphone world, is a process of obtaining the highest privilege in the operating system [10]. Root user in Android (and general Linux distributions), which has UID 0, is disabled in userland to limit malicious applications from gaining absolute control over the system. Application that has root privilege can virtually do anything, like unmounting file systems, killing processes, or running any arbitrary commands [14]. The common binary file that can be found in system partition of any rooted device is *su* file, which stands for "super user" or "switch user." Basically *su* has set-uid flag set and is always run as UID 0. Applications making use of root often try to execute this binary file, which is accompanied by a gatekeeper Android application (SuperSU.apk or SuperUser.apk) to escalate their privileges to the highest.

Rooting can be classified into 2 main types:

(1) Soft root, which is gaining root on a booted system: This technique is entirely software-based and can be achieved by exploiting vulnerabilities in the kernel, in a process running as root or even through a symbolic link [2]. Based on these vulnerabilities, many applications were created to provide users "one-click rooting" tools. Rooting achieved by soft root is usually temporary, especially in devices that have locked bootloader, as it checks the integrity of the system every time the device reboots. This requires additional work to persist in root privilege, namely, unlocking the bootloader and manipulating Recovery Mode. Attack vectors that support soft root will be discussed in the next section.

(2) Hard root, on the other hand, requires physical interaction with the device. The main idea is booting into Recovery Mode which is isolated from userland and performing *su* installation or replacing entire operating system with the new one that has "su" installed. On some devices, bootloaders are locked, which make the rooting process more difficult. Hard root does not suffer from temporary rooting problem like soft root does, as the bootloader is unlocked in advance, which makes it a permanent rooting technique.

We illustrate different types of rooting methods in Figure 2. All applications that are designed to use root privilege, regardless of benign or malicious intention, need to execute *su* file either through libraries or directly after obtaining permission from gatekeeper application (SuperUser.apk or SuperSU.apk) [15].

2.3. Android Hooking. When Android OS starts booting up, Zygote process will be started by *init.rc* script, or more specifically the */system/bin/app_process* native daemon [16]. Zygote is the parent of all application processes, and it is responsible for all of their forks. After all needed classes are loaded, Xposed framework [17] provides extended *app_process* to add an additional jar to the classpath and calls methods to act in Zygote context. This enables the possibility of hooking method calls and injecting custom code to manipulate the application's behaviors. For instance, Gajrani et al. [18] use this framework to hide their emulator from being detected by advanced malware. ARTDroid [19] is another work targeting ART Runtime that enables sandbox to analyze apps without being evaded. Similar to Xposed, this technique also requires root privilege to operate.

In this study, we mainly focus on Java hooking techniques, which utilize Xposed as an instrumental tool for rooting evasion. The work on Cydia Substrate for Hooking Native Code has been abandoned since 2013 [20]; therefore it is only applicable to Android 4.3 and older versions, which occupy less than 2% of the Android versions distribution [21].

2.4. The Risk of Rooting. As we discussed "one-click rooting" applications, they are usually transparent to users. In other words, users have no idea what happen beneath the GUI (Graphic User Interface) other than what they are interacting with. Therefore, malicious applications can exploit CVEs (Common Vulnerabilities and Exposures), especially in older devices, to cause severe impact on the system.

Dan Rosenberg has discovered the exploit of symbolic link attack by linking */data/* to *data/local/tmp* (readable-writable mode). By modifying */data/local.prop* file and changing *ro.kernel.qemu* to 1, it makes the ADB shell (Android Debug Bridge) switch user to root mode [22]. This vulnerability was fixed in Android 4.2.

In the case of applications that employ update-attack, once super user privilege is gained, these apps are able to update themselves and execute arbitrary commands to remount the system partition, make a copy, or even wipe out almost any data. Having said that, when malicious applications have root privilege, the system is already compromised. As an example, we refer to previous studies to prove and show that proposed security mechanisms based on the assumption

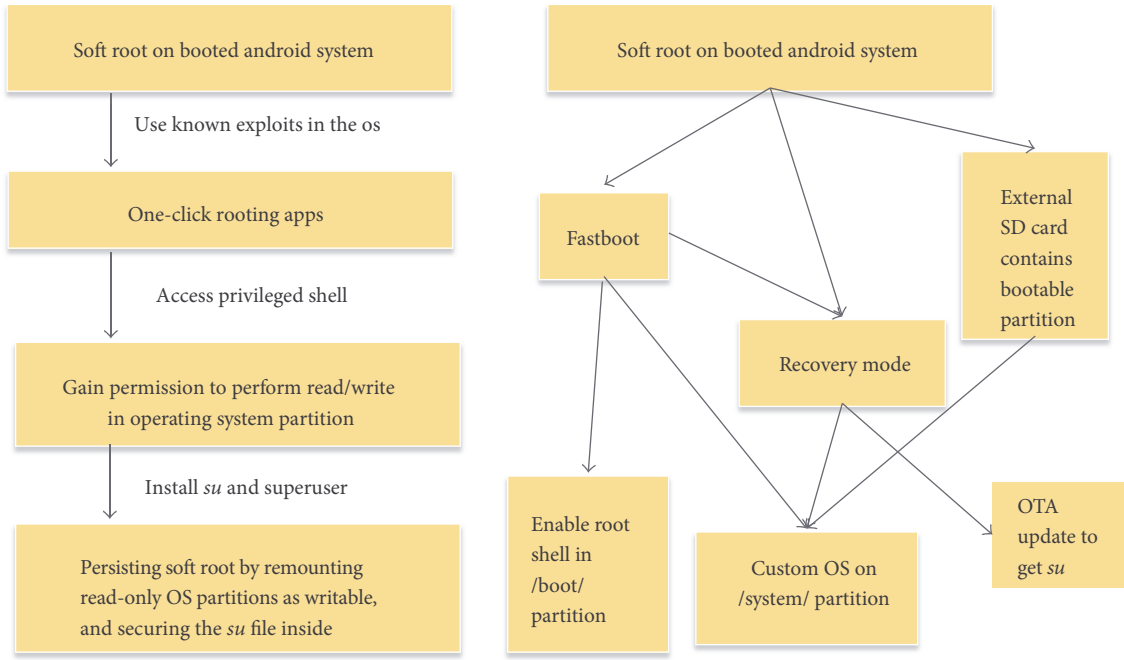

Figure 2: Rooting methods: soft root and hard root.

of a secure system are no longer achieved once the malware gains root power [23–25].

3. Related Works

In this section, we present some related works regarding rooting check/evasion and privilege escalation in Android device.

3.1. Papers and Study regarding Rooting Privilege Escalation. Sun et al. [14] conduct a study of 182 selected applications to identify current rooting detection methods. The authors present a comprehensive work of classifying rooting methods and propose different techniques for rooting detection and evasion. Seven categories for rooting detection have been studied carefully, namely, checking installed packages and files related to rooting, Build Tag, System Properties, Directory Permissions, running processes/services/tasks, and shell commands. To conclude the work, the authors state that the arms race between rooting detectors and evaders is an asymmetric competition that favors evaders. In other words, no matter how well-implemented detectors are, there will always be techniques to evade the rooting detection. The authors then suggest a reliable approach should come from the kernel realm, which is strongly immune to most attacks originated from higher layers of the ecosystem. While we agree with the conclusion that rooting always favors the evader compared to detector, we still believe that there is a huge demand for effective root checking applications. Most banking applications that we investigated so far employ at least some aforementioned checking techniques, in order to protect their applications from running in unsafe environment, which could be abused for further attacks.

We revisit all the rooting detection techniques mentioned in the paper and propose new methods proven to be working in larger scale of sample data. The finding we have come across is that Cydia Substrate [20] only supports Android 4.3 and older ones. That means the hooking techniques the authors have implemented will no longer work with 98.8% Android devices [21]. Moreover, as the authors mentioned in their study, RDAanalyzer is revised to evade newly discovered detection technique(s). We believe, without Cydia functioning, the only way to discover detection techniques in native code is reversing shared object files, which is time-consuming and really difficult to fully achieve. In Table 6, we have to perform memory analysis to collect the library files and keywords, and still we are not able to evade some banking/finance applications as described in Table 5. This finding encouraged us to develop a root checking application at both Java and native level in Section 4, which is not evaded by RootCloak [26] and other similar tools that employ Xposed framework [17].

Kim et al. [27] present an analysis on 76 popular financial apps in Korea to address their self-defense mechanisms, which are used to protect the application from rooted or tampered environment. The authors develop a tool named MERCIDroid to identify the causality between the *environment investigation* and the *execution termination*. MERCIDroid first looks for environment information providers, for example, *getPackageInfo()* and *File.exists()*, and uses investigating conditions like "the command name or file name is *su*" or "APK file related to the rooting is installed" to conclude that the device is rooted. Based on the SDMG (Self-Defense Mechanism Graph) constructed by the call graph, the tool then identifies the *execution terminator*, which is the API that triggers the app to stop (or display warning

dialog). The results show that there are 67 out of 73 apps that were successfully bypassed the rooting checks. The results, in our opinion, can be greatly enhanced by applying techniques mentioned in Section 4 of this paper. For instance, MERCIDroid only depends on few *environment information providers* to break the self-defense mechanisms; the "su" binary file name can be changed easily to avoid detection. The *System Properties* and *Build Tags* should also be added for the completeness of the check. Finally, regarding the work of modifying native code, the authors claimed to use IDA Pro [28] to bypass the Lib4 check but did not provide the detailed techniques. To the best of our knowledge, one of IDA plugins that is able to patch the binary file is IDA-Patcher [29], which is no longer in active development. Nevertheless, application secured by native code is hard to break; this suggests to us developing rooting detection mechanisms by using NDK (Section 4).

In the effort of limiting privilege escalation attack, a number of works have been proposed. Davi et al. [30] showed that it is possible to mount the attack at runtime through access control mechanism that allows transitive permission usage. Bugiel et al. [31–33] conduct a heuristic analysis of Android system to address collusion attacks and solve confused deputy problem, as well as presenting an extended monitoring mechanism for Android. Park et al. [34] present a solution called RGBDroid to protect the system even after root privilege has been granted to the malware. Xing et al. [35] reveal a privilege escalation named *Pileup* within updating logic that makes the malware acquire higher privileges even from signature permissions. Zhongyang et al. [36] demonstrate DroidAlarm to analyze capability leaks in Android malware. All of these studies so far have proven that there are still many works to be done to enhance the security of Android.

3.2. Google SafetyNet Attestation. Traditional rooting techniques are all about modifying the */system* partition to install custom script and gain root access. Chainfire, a senior developer from XDA, provides a different approach by modifying only the boot image in order to achieve root privilege; therefore the whole system remains intact [37]. Systemless rooting is considered to be a newer workaround for advanced users that want to use services that check for rooting without having to unroot their devices. As the newest update, this technique no longer works for apps that adopt Google SafetyNet (e.g., Android Pay) [38]. Google SafetyNet (or specifically SafetyNet Attestation), however, does not really target rooted environment. This tool provides a set of APIs that allows app developers to use them in their own apps to check for device safe state in general. Google uses a neutral term to describe it as "CTS" (Compatibility Test Suite). The results returned from the API are *basicIntegrity* and *ctsProfileMatch* which are the most important, as they indicate the integrity of the device. Another work has shown SafetyNet also targets rooting signs like "su" binary, and with proper modification, it can be deceived in API level [39].

3.3. Rooting Check Evasion. The OEMs nowadays start releasing their devices with unlocked bootloaders or unlockable

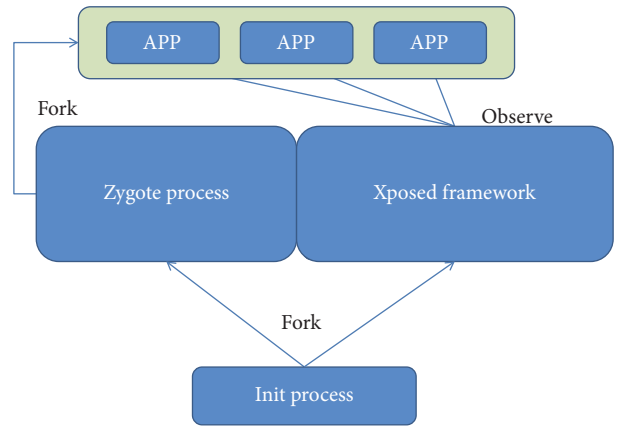

Figure 3: Xposed framework.

bootloaders. The restrictions of using only original OS have been loosen over the years as the trend of openness and free of use for mobile customers. Some OEMs like Google even give the users instruction to safely unlock the bootloader without bricking their phones. By doing these, users now can install their OS of choice for different purposes: extending features, installing new incompatible applications, and speeding up the device with overclocking; rooting plays an important role above all. Unfortunately, rooting is also a target of malicious software, which is the reason why some applications, ranging from banking to entertainment, at least try to detect it and refuse to run if the result is positive to rooting. Evasion techniques were born as the need of the users to bypass these checks, and they also accidentally help malware hide themselves.

One of the most important works on this subject is RootCloak [26]. This framework makes use of Xposed framework [17] and Cydia Substrate for Android [20] to bypass common checks. As described in Figure 3, Xposed modifies *app_process* and hooks apps accessed from the Xposed framework using *de.robv.android.xposed.installer/bin/XposedBridge.jar*. XposedBridge.jar hooking API in turn gets the Java ClassLoader object. Since ClassLoader contains the class to be called, it hooks the target method in ClassLoader by using the class name and method name that the user wants. RootCloak can virtually defeat any rooting check application at Java code, if the class and method name are given. With the prevalence of different reversing tools like enjarify [40], one can obtain the information without difficulties. RootCloak is developed as a module of the Xposed framework; it implements given APIs to intercept rooting check applications. As the example in the listing below, RootCloak is instrumented to hook into PackageManger for the list of installed applications:

```
findAndHookMethod(
"android.app.ApplicationPackageManager",
                    lpparam.classLoader,
        "getInstalledApplications",
        int.class, new XC_MethodHook()
```

By using this technique, RootCloak is able to bypass applications that check for rooting by intercepting ActivityManager, PackageManager, Runtime, and ProcessBuilder. For applications that check for specific keywords, filepath can be evaded by returning dummy results instead of sensitive strings like SuperSU, root, chainfire, and so on, which are typical signs of rooting. Some of them are described in Figure 6. Since the obsolete of Cydia Substrate, which only works in Android 4.3 and older, RootCloak no longer can evade root checking techniques that originate from native level, which is embedded in Android applications as shared object files (.so files). This downside of RootCloak makes it unable to bypass the check of several root checkers, especially advanced modules from banking and finance applications. The explanation of how these modules work and the analyses of the banking and finance applications are discussed in Sections 5 and 6.

4. Root Checking Methods

Despite the possibility of being evaded by the same approach we soon present in Section 5, we still witness the existence of root checking techniques in critical applications, which *is proven to be needed by Google Report and several banking/finance applications we have analyzed*. While many applications do employ Native Development Kit to bypass API hooking, they are in short of checkpoints and fall into this evasion. As we previously mentioned, 10 out of 18 banking and finance applications only check for the existence of *su* binary file by its name, so the evasion is just as simple as changing the name of this binary file by rebuilding it. In order to encounter this technique, we propose regular expression matching for file names and process and service names and also check the hash value of each file (if they are readable) against our database, which is the collection of multiple rooting apps and binaries hash values. We apply this kind of check almost everywhere in each category below to guarantee the effectiveness of detection.

In this section, we revisit the work of checking rooted device mentioned in [14] and 92 rooting check applications we have collected from Google Play Store, as well as proposing some improvements and directions to enhance the effectiveness of this work. For the sake of completeness, we present all available techniques collected from our study. The Native Development Kit (NDK) is a set of tools provided by Google that allows us to use C and C++ code with Android and provides platform libraries you can use to manage native activities and access physical device components, such as sensors and touch input. The NDK may not be appropriate for many novice Android programmers who need to use only Java code and framework APIs to develop their apps. Even some of Android apps can be evaded by API (SDK) hooking; we realize that RootCloak is still not ready to hook into native code at the time of this writing. While most of root checking features in Java can be reimplemented in native code, we decide to discuss both at the same time and point out the differences if necessary. To clarify the work, we refer to "keywords," "processes," "daemons," "package names," and "directories" in Tables 1, 2, and 3.

Table 1: Keywords used for rooting check.

Keywords	Use case
daemonsu superuser supersu kinguser kingroot busybox	Rooting-related binaries and daemons: these keywords are typical in rooted devices
rootcloak xposed chainfire	Keywords related to evasion apps: to evade rooting check at Java code, RootCloak uses Xposed framework to perform interception with API calls
titanium greenify stericson kerneladiutor	Keywords related to apps that require rooted device

Table 2: Package names used for rooting check.

Package names	Use case
com.jrummy.root.browserfree	Directory browsing
rootcloak xposed chainfire	Mount internal to external storage
com.oasisfeng.greenifiy	Android utility tool
com.jrummy.apps.build.prop.editor com.grarak.kerneladiutor org.namelessrom.devicecontrol com.jumobile.manager.systemapp	Modifying settings, System Properties, scanning device partition
stericson.busybox	Linux commands utility
de.robv.adnroid.xposed.installer com.devadvance.rootcloak com.devadvance.rootcloakplus	Root hiding

Table 3: Directories used for rooting check.

Directories	Use case
/system/bin /system/xbin /sbin /data/data /system/usr /system/bin/.ext	Likely contain *su* binary
/system/app /data/app /data/dalvik-cache	APK files and cache

Within the scope of this paper we only provide some of the most popular keywords for the sake of intuitive demonstration. While we still apply static terms to check for rooting, different tactics will be discussed in detail for each technique to enhance the accuracy of the detection.

Check Installed Packages. *PackageManager* in Android is responsible for manipulating all installed applications. By querying all installed apps, we are be able to get the list of package names that could be useful for the detection. Different types of apps are used for the check.

(1) One-Click Rooting Apps. Such applications are not allowed in Google Play Store due to the danger of exploitation.

(2) Root Manager Apps. To name but a few, there are SuperSU, SuperUser, Kingroot, and Kingo apps.

(3) Root Apps. These are the most popular with hundreds of applications in Google Play Store and in the Wild, due to the demand of users.

We have collected these applications, as well as the MD5 hash values and their package names. Devices that have one of them installed are obviously susceptible to rooting.

This can also be achieved by executing *pm list packages* command.

Check Existence of Files. Rooting counts on different files to function properly: su binary and rooting-related APK files (SuperSU.apk or SuperUser.apk and applications that require root privilege to operate). These APK files will not be necessary after installation process, but they are kept by default unless users delete them intentionally. Checking the existence of certain files is considered to be an important factor of rooting detection. Our tool tries to check the existence of those files in different directories: */system/bin/* and */system/xbin/* for su binary; */system/app* and */data/app* for APK files. Android SDK and NDK provide *File* (at Java level) and *(*FILE) popen* (native code) to carry out the work.

One common weakness of root checking applications is that they tend to use static keywords, which makes the search less flexible. In order to improve the accuracy, we apply the regex (regular expression) to maximize the search range. For example, the regex of *filename_.* _[123]\\.apk* will match any file that looks like these: filename_aa_1.apk or filename_bb_3.apk.

In native code, this can be achieved by JNI (Java Native Interface) code *FILE* file = fopen("file_path")*.

Check Processes, Services, and Tasks. As of Lollipop (API level 21), *getRunningTasks* API was no longer available for third-party applications due to personal information leakage to the caller. Therefore, we only evaluate running processes and services in *ActivityManager*. We use a list of collected keywords related to rooting in order to perform the matching with process and service names. Kingroot, SuperSU, and Kingo are the most popular tools (Windows and Android applications) appearing on rooted devices; ones having such keywords are likely to be rooted. We use *getSystemService* to retrieve a *ActivityManager* and interact with system state and get the list of running processes *(RunningAppProcessInfo)* and services *(RunningServiceInfo)*.

As Android is Linux-based, *ps -aux* command can also list processes and services available.

Check Shell Commands Execution. Android OS has some legacy commands inherited from Linux; some of them can be executed as a normal user; others can only be executed as root. We try different commands and investigate the returned values to see if the device is rooted or not based on different factors.

Determine the existence of some files or text of interest: by executing *su*, *pm list packages |grep <a_keyword>*, *ls <a_directory>*, *cat <full_path_to_a_file>*, *ps -x*, and so on, the ouput can be a full path which led to *su* binary, suspicious process(es), text strings containing keywords in Table 1, or package names in Table 2.

In native code, one can use *system(command_to_be_run)*. This also covers the previous cases of checking processes/services and listing package names.

Check Build Tag and System Properties. Rooted device contains different values of Build Tag and System Properties when the device has custom built OS. If *android.os.Build.TAGS* is "test-keys" instead of "release-keys," this indicates an unofficial build. With System Properties, we can either use SDK APIs for checking the rooting, *readSystemProperty("ro.debuggable")* and *readSystemProperty("ro.secure")*, or execute *getprop |grep -wE "ro.debuggable |ro.secure"* as a normal user. If the returned values are 1 and 0, consecutively, that means an ADB shell will be able to run as root user on the device shell.

Directory Permissions. Rooting sometimes changes permissions (read, write, execute, and list) in certain directories. Table 3 can be used in combination with Java or native code to compare the differences with unrooted devices. For example, */data/* and its subdirectories by default are immutable to read, write, and execute by default. If some rooting applications accidentally or intentionally change permissions in these locations to global read/write, normal user and apps will be able to perform operations in them. So far, we did not find any rooting signs based on this check.

The simple commands like *ls -l* may reveal the current status of certain file or directory.

The work flow of our rooting check application targeting Android 4.4 and later ones is provided in Figure 4.

5. Techniques to Bypass Current Root Checking Apps

Based on the formative study on Section 2 about Android architecture and rooting methods, we realize the correlation between the methods (APIs) and the objects can be intercepted. The reason is that the API calls of Android SDK are made from application layer to the framework (see Figure 1), which can be modified at will when the OS was compromised. For instance, a rooting check application may utilize *PackageManager* to check the package names of all installed applications by using *getInstalledApplications* API.

By analyzing the decompiled source code of different rooting check applications described in Section 6, we have a sufficient list of related APIs that check for the signs of rooting (Table 4). Moreover, the study on rooting check applications in Section 4 gave us a hint to target several rooting-related objects, in the form of files ("su", "SuperSU.apk", top root apps in Google Play, etc.) or in the form of distinct keywords ("chainfire", "superuser", "Runtime.exec()", etc.). We consequently created a hooking tool which targets current root checking applications. To this end, we realize that most

Table 4: APIs for rooting check in 89 applications.

Class name	Number of apps using these APIs	Fraction
android.app.ApplicationPackageManager.getInstalledPackages	1	0.61%
android.app.ApplicationPackageManager.getInstalledApplications	0	0%
android.app.ApplicationPackageManager.getPackageInfo	8	4.85%
android.app.ApplicationPackageManager.getApplicationInfo	13	7.88%
android.app.ActivityManager.getRunningServices	1	0.61%
android.app.ActivityManager.getRunningTasks	0	0%
android.app.ActivityManager.getRunningAppProcesses	12	7.27%
java.lang.Runtime.exec	49	29.70%
java.lang.ProcessBuilder	25	15.15%
java.io.File	52	31.52%
java.lang.Class.forName	4	2.42%

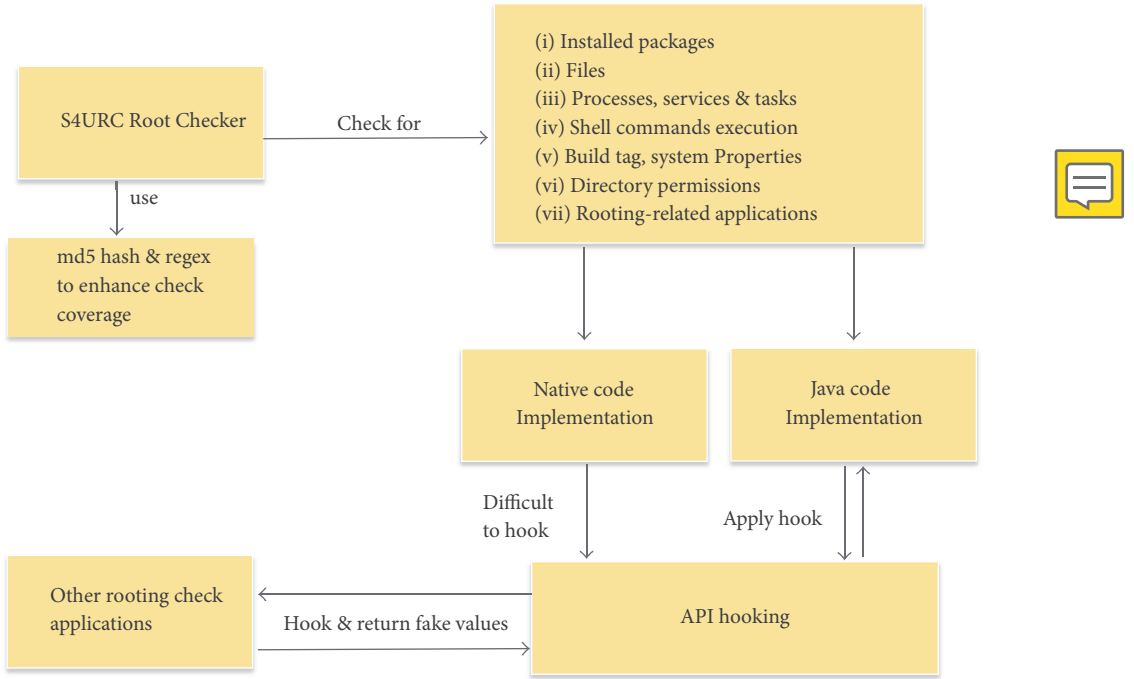

Figure 4: Rooting check and evasion.

of them are poorly designed and vulnerable to evasion technique like Java APIs hooking. For instance, we manually tested the top 92 applications in Play Store by providing the keyword "root checker" (dataset 1-a in Section 6). 89 applications have been evaded, 2 applications crashed unexpectedly due to incompatible APIs, and only 1 application was not evaded.

We continue to follow the case of 18 banking and finance applications that applied more advanced rooting check techniques in native code [41]. Java APIs hooking fails completely to evade these apps even though we have tried to customize the hooking tool as much as possible. We perform dynamic analysis on the memory of each banking applications by reading */proc/$pid/mem* while running these apps on the rooted device. By doing this way, we are able to collect interested strings related to rooting, as described in Table 6. With trial and error, we managed to evade 10 banking applications by modifying the *su* binary name into a different name (i.e., susu) or adjusting the names of APK files to avoid the file name check. By executing that *su* file with its new name, root privilege is still granted. In other words, banking and finance applications can still be evaded.

6. Result Analysis

In this section, we continue to elaborate the rooting problems by evaluating multiple datasets that we have been collecting over the years, ranging from malware in the Wild to rooting-related apps and benign apps in Google Play Store.

We first describe our datasets as follows.

(1) Applications in Google Play Store. All of them were downloaded from December 2016 to February 2017.

Table 5: Evading 18 banking and finance applications.

Index	Name	Evaded
1	Banking App 1	Yes
2	Banking App 2	Yes
3	Banking App 3	Yes
4	Banking App 4	Yes
5	Banking App 5	Yes
6	Finance App 1	Yes
7	Finance App 2	Yes
8	Finance App 3	Yes
9	Finance App 4	Yes
10	Finance App 5	Yes
11	Banking App 6	No
12	Banking App 7	No
13	Banking App 8	No
14	Finance App 6	No
15	Government 1	No
16	Government 2	No
17	Banking App 9	No
18	Banking App 10	No

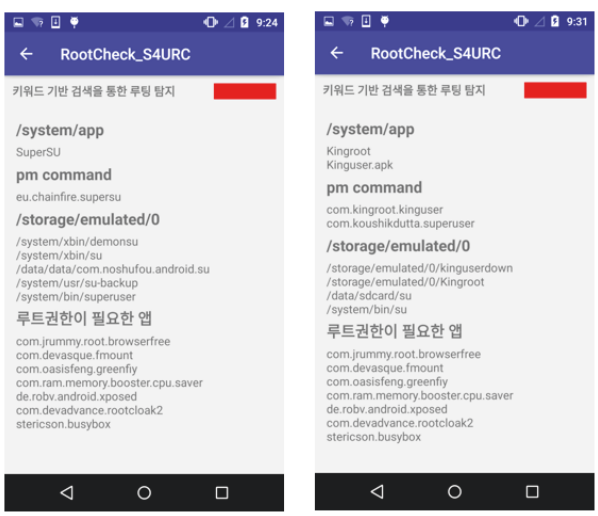

Figure 5: S4URC Root Checker.

(a) Root Checking Apps. They include 92 independent (free) root checking apps and 18 root checking modules from banking/finance apps.

(b) Root Apps. Top 35 apps that request using root privilege in Google Play Store; each has more than 1,000,000 users. These apps enable users to access rich features that are not available in the original build from OEMs (Original Equipment Manufacturers).

(c) Others. They include 7200 benign applications from Google Play Store, which are randomly selected from several categories of Google Play (sport, game, education, entertaining, etc.) to avoid bias. We want to use these apps to measure the fraction of apps related to rooting.

(2) Malicious Applications in the Wild. 21061 malware types are mostly acquired from VirusShare (2013-2014) [42] with a small portion (roughly 1%) from Contangio Minidump [43] manually. We use this dataset to evaluate the popularity of rooting-related malware, compared with dataset 1-b and 1-c from Google Play Store.

6.1. Rooting Check Applications. We chose the top results that returned the search keyword "root checker." 92 rooting check applications in Google Play Store were downloaded and manually tested with and without RootCloak evasion. 89 applications have been evaded by RootCloak, 2 applications were crashed when launching, and only 1 application escaped the evasion of RootCloak. As we continued reversing each application, we found out that the app can avoid the evasion that employed the check module in its native code, thus being immune to SDK API hooking. To get an insight into these apps, we carefully checked each of the 89 applications by using Xposed framework to hook into 11 classes as described in Table 4 and Figure 7. Most of them try to check the existence of rooting-related applications *(getApplicationInfo)*, files *(java.io.File)*, and processes *(java.lang.ProcessBuilder)* and also try to execute shell commands through *java.lang.Runtime.exec* class. As shared objects (developed in native code) analysis is out of the scope of this study and these applications are already defeated by Java API hooking, we did not try to investigate the their source code any further.

6.2. Banking and Finance Applications in Korea. Rooting check is also embedded as a module inside banking and finance application to avoid tampered environment. We selectively tested RootCloak with 18 applications; all of them were able to circumvent RootCloak promptly and refused to run in rooted environment. We tried to customize RootCloak with Xposed as much as possible but these banking apps employ their checking modules by native code so it is impossible to evade them by SDK API hooking. By trial and error, we eventually found a way to avoid detection just by changing the *su* binary file name into different name. 10 out of 18 applications can be evaded by this way (Table 5). As the disclosure of investigated participants may reveal their sensitive information, which should be strictly avoided, *we decided to hide all real names of 18 banking/finance applications* after receiving 2 responses from the banking contact points. Rightful parties will be provided with necessary data if needed. These apps only use native code to check the existence of *su* file, which is easy to bypass without the need of intercepting native calls. By changing the file name, we were still able to execute *su* file successfully and gained root privilege. But this method will no longer work if the device resets, as all processes related to rooting will be named consequentially terminated without being able to recover due to the change of *su* file. This issue can still be addressed by recompiling *su* and *SuperSU* app altogether to update the

Table 6: Root checking in native library.

Package name	Library name	Root detection
Finance App 1	libXAS_jni.so	/sbin/su /system/bin/su /system/sbin/su /system/xbin/su /system/xbin/sudo /system/app/superuser.apk /system/app/UnRoot.apk /system/app/Nakup.apk /data/data/com.noshufou.android.su /data/data/com/ajantech.app/UnRoot /data/app/com.noshufou.android.su-2.apk
Government 1	libsmartmedic.so	Store rooting_pattern information into smartmedic1.db file
Security App 1	libEngineManager.so	native.startRootCheck
Banking App 1	libBengine.so	Check */proc/%d/cmdline* for su
Banking App 2	libap1.7.8.so	Check su files Check Telephony status Check Wifi status (including MAC address) Check for noshufou and supersu
Finance App 2	libap1.7.2.so	Check for noshufou and supersu

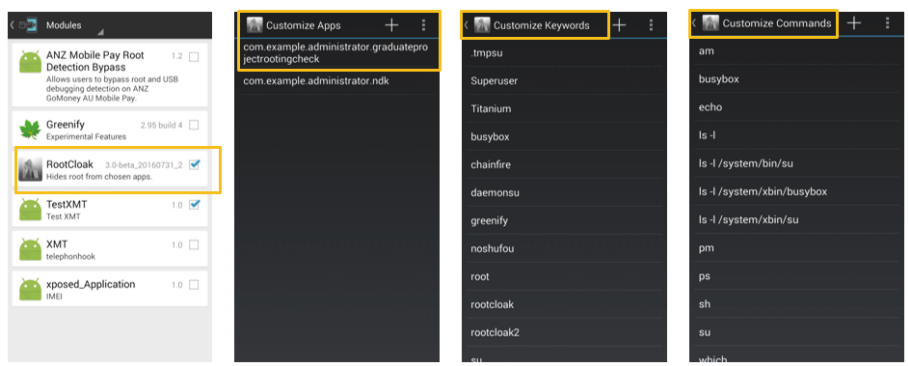

Figure 6: Evading root checker by RootCloak and Xposed.

new binary name. We continued our work by analyzing the native libraries of some finance and banking applications to have better understanding of what these apps are doing beneath those SO files. Their SO files are listed in Table 6. Once again, for security matter, we do not disclose their identities. It turns out that these apps also check for the existence of package names, which is fairly inefficient as they are all static and can be manipulated to bypass the check. We did not investigate the case any further because it is absolutely feasible to customize the rooting process with new *su* name to evade these banking apps. Customized root apps with the new package names and binaries are also able to operate with the new *su*. For what it is worth, each of these banking/finance apps has more than 1 million users, which makes them attracted to attackers.

6.3. Other Analyses. To complete our study, we performed some basic analyses with the rest of our collected applications. We divided them into 3 groups (namely, datasets 1-b, 1-c, and 2) as in Table 7 and used predefined keywords to scan through each application. Our goal is to determine the prevalence of rooting-related applications in Google Play Store, as well as of malware from the Wild. The results surprised us with the frequency of using keywords related to rooting in benign applications which are much higher (there are only 7200 benign applications, compared to more than 21,000

Table 7: Scanning rooting-related applications by popular keywords.

Keyword	Root apps	Benign apps	Malware
chainfire	3	9	13
root	28	3740	1348
libsuperuser	0	2	0
supersu	7	20	7
superuser	12	57	220
shell	10	367	172
Runtime	3	426	63
sudo	0	443	148

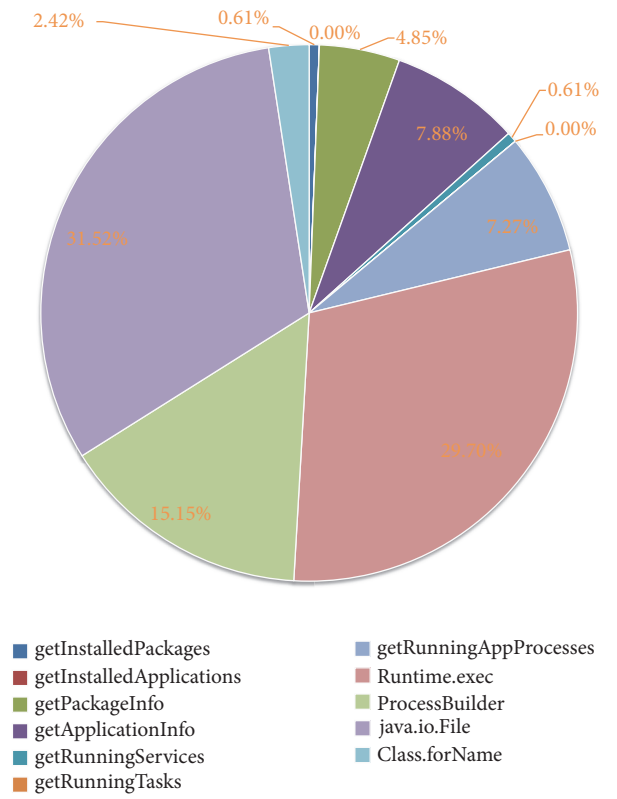

Figure 7: Approximate Pie Chart of 11 prevalent API classes used in root checking apps.

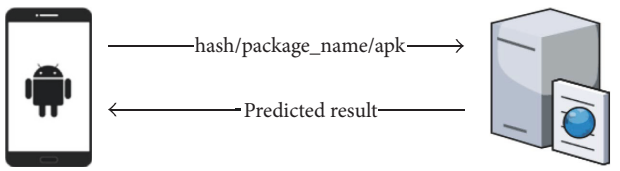

Figure 8: Root app download prediction.

malware types). Note that dataset 1-c (7200 benign apps) is more recent (2016-2017) than dataset 2 (21061 malware types collected from 2013-2014) but is less in number. That means the rooting-related applications have become more popular over the years. While the list of keywords is not complete and there are too many factors to be evaluated (e.g., some of benign apps can also be rooting check apps by accident, which affect the count in Table 7), we only consider this analysis as a reference to measure the fraction of rooting-related applications and do not proceed any further.

6.4. Cloud-Based Root App Download Prediction. Based on the signatures generated from malware and root app repositories, we extend our work to the cloud-based prediction of root app downloaded to mobile device. Whenever a user downloads and installs any APK file in Play Store or from the Wild, it will trigger our application to send some basic information like hash value and package name to cloud system, which in turn matches with the database and returns the alert if that is a root app, as in Figure 8. In our opinion, it can help prevent the inadvertent rooting from one-click rooting applications and protect the device from any root app in general. In our future work, this will be integrated with our currently published rooting check application (in Figure 5) [12].

7. Conclusion

In this paper, we have discussed different aspects of Android rooting, including current trend of Android users, evasion techniques, and countermeasures. As an open ecosystem, Android attracts a lot of attention from the good and the bad and may result in huge impact due to its large userbase once vulnerabilities are introduced. The trend of rooting is inevitable as it gives the users full control over the devices they own. But, still, the large portion of users are not aware of potential danger running in background. Sooner or later, detection methods proposed in this study will be evaded since the developers are able to customize their ROM and boot image at will, which could completely eliminate the possibility of accurate checks. But, in the meantime, we still believe such detection methods are necessarily required to provide a certain level of confident security to the users. Rooting evasion and rooting detection are always on the race that favors the evader than the detector. And in the researchers' point of view security check for rooting must be adopted at kernel level, which heavily depends on the OEMs and Google to avoid such evasion techniques effectively.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2017-2012-0-00646) supervised by the IITP (Institute for Information & Communications Technology Promotion). This work was supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (no. 2016-0-00078, Cloud Based Security Intelligence Technology Development for the Customized Security Service Provisioning).

References

[1] IDC, "Android market share," 2017, http://www.idc.com/promo/smartphone-market-share/os.

[2] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android Hacker's Handbook*, John Wiley & Sons, 2014.

[3] E. Nikolay, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*, No Starch Press, 2014.

[4] J. Levin, Android Internals: The Confectioner's Cookbook (VOL1 The Power User's View), 2016.

[5] Y. Shao, X. Luo, and C. Qian, "RootGuard: protecting rooted android phones," *The Computer Journal*, vol. 47, no. 6, Article ID 6838907, pp. 32–40, 2014.

[6] Google, "Android security report 2014," 2015, https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf.

[7] Google, "Android security report 2015," 2016, https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf.

[8] Google, "Android security report 2016," 2017, https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf.

[9] Chainfire, "SuperSU," 2017, http://www.supersu.com.

[10] T. Vidas, C. Zhang, and N. Christin, "Toward a general collection methodology for Android devices," *Digital Investigation*, vol. 8, pp. S14–S24, 2011.

[11] T. Jason, *XDA Developers' Android Hacker's Toolkit: The Complete Guide to Rooting, ROMs and Theming*, John Wiley & Sons, 2012.

[12] L. Nguyen-Vu, "S4URC Advanced Root Checker," 2017, https://play.google.com/store/apps/details?id=ssu.cnsl.S4URC.rootchecker.

[13] Android Official Website, "Android security architecture," 2017, https://source.android.com/security.

[14] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015*, pp. 3–14, Denver, Colo, USA.

[15] Chainfire, "How-to SU," 2017, https://su.chainfire.eu.

[16] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*, O'Reilly Media, 2013.

[17] V. Costamagna and F. Bergadano, "HOOKDROID: Dalvik dynamic instrumentation for security analytics," *International Journal on Information Technologies & Security*, vol. 8, no. 3, 2016.

[18] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti, "A robust dynamic analysis system preventing SandBox detection by android malware," in *Proceedings of the 8th International Conference on Security of Information and Networks, SIN 2015*, Sochi, Russia, September 2015.

[19] V. Costamagna and C. Zheng, "ARTDroid: A virtual-method hooking framework on android ART runtime," in *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security, IMPS 2016*, pp. 20–28, London, UK.

[20] F. Sierra and A. Ramirez, "Defending Your Android App," in *Proceedings of the the 4th Annual ACM Conference*, pp. 29–34, Chicago, Ill, USA, September 2015.

[21] Android Developer, "Android platform versions," 2017, https://developer.android.com/about/dashboards/index.html.

[22] H. Zhang, D. She, and Z. Qian, "Android root and its providers: a double-edged sword," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015*, pp. 1093–1104, Denver, Colo, USA, October 2015.

[23] L. Nguyen-Vu and J. Souhwan, "DroidSecure: a technique to mitigate privilege escalation in android application," *Journal of The Korea Institute of Information Security & Cryptology*, vol. 26, 2016.

[24] L. Nguyen-Vu, J. Park, N.-T. Chau, and S. Jung, "Signing key leak detection in Google Play Store," in *Proceedings of the 30th International Conference on Information Networking, ICOIN 2016*, pp. 13–16, Kota Kinabalu, Malaysia, January 2016.

[25] C. Mulliner, W. Robertson, and E. Kirda, "VirtualSwindle: an automated attack against in-app billing on Android," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS 2014*, pp. 459–470, Kyoto, Japan, June 2014.

[26] N. Evans, A. Benameur, and Y. Shen, "All your root checks are belong to Us: the sad state of Root detection," in *Proceedings of the 13th ACM International Symposium on Mobility Management and Wireless Access, MobiWac 2015*, pp. 81–88, Cancun, Mexico, November 2015.

[27] T. Kim, H. Ha, S. Choi, J. Jung, and B. Chun, "Breaking Ad-hoc runtime integrity protection mechanisms in android financial apps," in *Proceedings of the the 2017 ACM*, pp. 179–192, Abu Dhabi, United Arab Emirates, April 2017.

[28] HEX-RAY, "IDA Pro," 2017, https://www.hex-rays.com.

[29] P. Kacherginsky, "IDA Patcher," 2014, https://github.com/iphelix/ida-patcher.

[30] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *International Conference on Information Security*, Lecture Notes in Computer Science, pp. 346–360, Springer, Berlin, Germany, 2010.

[31] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS '12)*, vol. 17, San Diego, Calif, USA, 2012.

[32] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: a new android evolution to mitigate privilege escalation attacks," Tech. Rep., Technische Universitt Darmstadt, Darmstadt, Germany, 2011.

[33] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Poster: the quest for security against privilege escalation attacks on android," in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 741–744, Chicago, Ill, USA, October 2011.

[34] Y. Park, C. H. Lee, C. Lee, J. H. Lim, M. Park, and S.-J. Cho, "RGBDroid: a novel response-based approach to android privilege escalation attacks," in *Proceedings of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, vol. 12, San Jose, Calif, USA, 2012.

[35] L. Xing, X. Pan, R. Wang, K. Yuan, and X. F. Wang, "Upgrading your Android, elevating my malware: privilege escalation through mobile OS updating," in *Proceedings of the 35th IEEE Symposium on Security and Privacy, SP 2014*, pp. 393–408, San Jose, Calif, USA, May 2014.

[36] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS 2013*, pp. 353–358, Hangzhou, China, May 2013.

[37] P. C. Abhishek, "Analysing the vulnerability exploitation in Android with the device-mapper-verity (dm-verity)," in *Proceedings of the Symposium on Applied Computing*, pp. 576-577, Marrakech, Morocco, April 2017.

[38] M. Schölzel, E. Eren, K.-O. Detken, and L. Schwenke, "Monitoring Android devices by using events and metadata," *International Journal of Computing*, 2016.

[39] J. Kozyrakis, "Inside Google SafetyNet," 2017, https://koz.io/tags/safetynet.

[40] Google, "Enjarify - Android Reversing Tool," 2017, https://github.com/google/enjarify.

[41] G. Dana, M. Nigmatullin, and R. Bierens, "Jailbreak/Root Detection Evasion Study on iOS and Android," Research Report, University of Amsterdam, 2016.

[42] J.-M. Roberts, "VirusShare," 2017, https://virusshare.com.

[43] M. Parkour, "Contangio Minidump," 2017, http://contagiominidump.blogspot.com.